Article

Not peer-reviewed version

XTorch: A High-Performance C++ Framework for Deep Learning Training

Kamran Saberifard *

Posted Date: 7 July 2025

doi: 10.20944/preprints202507.0540.v1

Keywords: XTorch; C++ Deep Learning; High-Performance Computing (HPC); LibTorch; Data Loading; Performance Optimization; Generative Adversarial Networks (GANs)

 doi:10.20944/preprints202507.0540.v1

Article

XTorch: A High-Performance C++ Framework for Deep Learning Training

Kamran Saberifard

Independent Researcher; kamisaberi@gmail.com

Abstract

The deep learning ecosystem is predominantly driven by high-level Python frameworks like PyTorch and TensorFlow, which offer exceptional flexibility and ease of use. However, the reliance on a Python front-end can introduce significant performance overhead, particularly in data-intensive training pipelines, often necessitating multi-GPU setups to achieve acceptable training times. This paper introduces **XTorch**, a high-level C++ deep learning framework built atop LibTorch, designed to bridge the gap between Python's usability and C++'s raw performance. XTorch provides a familiar API for datasets, transforms, and models while eliminating Python-related bottlenecks. We demonstrate its efficacy by training a Deep Convolutional Generative Adversarial Network (DCGAN) on the CelebA dataset. Our results show that XTorch, running on a **single NVIDIA RTX 3090 GPU**, completes a 5-epoch training run in **219 seconds**. This represents a **37% speedup** over a standard PyTorch implementation which required **350 seconds** using **two RTX 3090 GPUs** with `DataParallel`. This work validates that a native C++ framework can not only match but significantly outperform common multi-GPU Python setups, offering a compelling case for reducing hardware costs and accelerating research and deployment.

Keywords: XTorch; C++ deep learning; High-Performance Computing (HPC); LibTorch; data loading; performance optimization; Generative Adversarial Networks (GANs)

---

1. Introduction

The last decade has seen an explosion in the adoption of deep learning, largely enabled by frameworks like PyTorch [1] and TensorFlow [2]. Their Python APIs have democratized AI development, making it accessible to a wide audience. While these frameworks perform core computations using highly optimized C++/CUDA backends, the "glue" logic—data loading, preprocessing, and the training loop itself—is typically executed in Python. This architecture, while flexible, introduces inherent performance limitations due to Python's Global Interpreter Lock (GIL) and general interpreter overhead.

In many real-world scenarios, especially those involving large datasets and complex image augmentations, the data loading and preprocessing pipeline becomes the primary bottleneck, leaving expensive GPU hardware underutilized. The common solution is to scale horizontally, either by increasing the number of CPU workers for the data loader or by using multiple GPUs. While effective, this approach increases hardware complexity, energy consumption, and operational costs.

We argue that for high-performance applications, a framework that operates entirely within a compiled C++ environment offers a superior alternative. By removing the Python front-end, we can eliminate interpreter overhead and implement more efficient, low-level data handling mechanisms using C++ concurrency primitives.

This paper introduces **XTorch**, a framework designed to provide a Python-like development experience directly in C++. It offers high-level abstractions for common deep learning components, including a model zoo, data transformation pipelines, and a highly optimized data loader. Our primary contribution is to demonstrate empirically that this C++-native approach can lead to dramatic

 doi:10.20944/preprints202507.0540.v1

performance improvements. By benchmarking a DCGAN [3] training task on the CelebA dataset [4], we show that XTorch on a single GPU can outperform a standard dual-GPU PyTorch implementation, effectively halving the hardware requirement while simultaneously reducing training time.

2. The XTorch Framework

XTorch is designed with two core principles: **performance-first** and **developer-friendliness**. It leverages the power of LibTorch for its tensor operations and automatic differentiation engine but provides a higher-level, more expressive API for building and training models.

2.1. Architecture

The XTorch library is structured into several key modules, mirroring the familiar organization of `torchvision`:

- `xt::models`: A collection of pre-implemented, standard neural network architectures. The DCGAN Generator and Discriminator used in our experiment are part of this module. This allows for rapid prototyping without needing to redefine common models from scratch.
- `xt::datasets`: C++ classes for interfacing with popular datasets. The `xt::datasets::CelebA` class handles the parsing of the dataset directory and attribute files, abstracting away the file I/O boilerplate.
- `xt::transforms`: A suite of data preprocessing and augmentation modules that mimic `torchvision.transforms`. The `Compose` class allows users to chain transformations like `Resize`, `CenterCrop`, and `Normalize` into a sequential pipeline.
- `xt::dataloaders`: This is the cornerstone of XTorch's performance. We provide an `ExtendedDataLoader` that is a ground-up C++ implementation of a parallel data loader.

2.2. The ExtendedDataLoader

The primary performance bottleneck in Python-based training is often the `torch.utils.data.DataLoader`. While it supports multi-processing via the `num_workers` argument, it suffers from the overhead of Inter-Process Communication (IPC) for transferring preprocessed data back to the main process.

The `xt::dataloaders::ExtendedDataLoader` bypasses this issue by using a C++ multi-threaded architecture.

1. **Multi-threaded Prefetching:** The data loader spawns a pool of C++ worker threads. Each thread independently fetches, decodes, and transforms a batch of data.
2. **Shared Memory Queue:** The processed tensor batches are placed into a concurrent, thread-safe queue. This avoids the costly serialization/deserialization and IPC overhead inherent in Python's multi-processing approach.
3. **Maximized GPU Saturation:** The main training loop thread simply dequeues a ready batch and moves it to the target device, ensuring the GPU is fed a continuous stream of data.

This design ensures that data preprocessing occurs in parallel with GPU computation, effectively hiding the data loading latency and maximizing GPU utilization.

3. Experimental Setup

To provide a fair and compelling comparison, we configured an experiment using a widely recognized model and dataset.

- **Hardware:**
  - CPU: AMD Ryzen 9 5950X (16-core)
  - GPU: NVIDIA RTX 3090 (24GB VRAM)
  - RAM: 64GB DDR4
- **Software:**
  - OS: Ubuntu 20.04

 doi:10.20944/preprints202507.0540.v1

  - CUDA Toolkit: 11.6
  - PyTorch / LibTorch: 1.12.1 (cxx11 ABI)
  - Compiler: g++ 9.4.0
- **Dataset:** CelebFaces Attributes (CelebA) dataset.
- **Model:** Deep Convolutional Generative Adversarial Network (DCGAN).
- **Training Parameters:** Epochs: 5, Batch Size: 128, Optimizer: Adam (`lr=0.0002`, `beta1=0.5`), Image Size: 64x64.

Benchmark Implementations:

1. **PyTorch Baseline:** The official PyTorch DCGAN example, modified to use two RTX 3090 GPUs via `torch.nn.DataParallel`. The `DataLoader` was configured with `num_workers=8`.
2. **XTorch Implementation:** The C++ code provided in Appendix A, run on a **single RTX 3090 GPU**. The `ExtendedDataLoader` was configured with `num_workers=2`.

4. Results and Analysis

The total time taken to complete 5 epochs of training is summarized in Table 1.

Table 1. Benchmark Results for 5-epoch DCGAN Training on CelebA.

Framework	GPU Configuration	Total Time (seconds)
PyTorch	2 x RTX 3090 (DataParallel)	350 s
XTorch	**1 x RTX 3090**	**219 s**

The results are unambiguous. The XTorch implementation on a single GPU was **131 seconds (37.4%) faster** than the dual-GPU PyTorch baseline. This significant performance differential can be attributed to three main factors:

1. **Elimination of Data Loading Bottlenecks:** The primary contributor to the speedup is the efficiency of the C++ `ExtendedDataLoader`.
2. **No Python Interpreter Overhead:** The main training loop in C++ is compiled to highly efficient machine code, avoiding the accumulated overhead of a Python-based loop.
3. **Inefficiency of `DataParallel`:** The single-GPU XTorch implementation completely avoids the scatter/gather overhead inherent in the `DataParallel` module.

5. Conclusions and Future Work

This paper introduced XTorch, a high-performance C++ deep learning framework. Through a direct comparison with a standard multi-GPU PyTorch setup, we have demonstrated that a C++-native approach can provide substantial performance benefits, cutting training time by over 37% while halving the required GPU hardware. The results challenge the conventional wisdom that C++ is only suitable for inference deployment. Future work will focus on expanding the XTorch library and implementing a C++-native equivalent of `DistributedDataParallel` for efficient multi-node training.

Appendix A. XTorch DCGAN Training Source Code

This appendix contains the complete C++ source code used for the XTorch benchmark.

```cpp
#include <xtorch/xtorch.h>
#include <torch/torch.h>
#include <iostream>
#include <chrono>
#include <numeric>

using namespace std;

int main()
{
```

 doi:10.20944/preprints202507.0540.v1


```
    try
    {
        // Hyperparameters
        const int nz = 100; // Size of latent vector
        const int ngf = 64; // Size of feature maps in generator
        const int ndf = 64; // Size of feature maps in discriminator
        const int nc = 3; // Number of channels
        const int num_epochs = 5;
        const int batch_size = 128;
        const double lr = 0.0002;
        const double beta1 = 0.5;
        const vector<int64_t> image_size = {64, 64};

        // Device
        torch::Device device(torch::cuda::is_available() ? torch::kCUDA : torch::kCPU);
        std::cout << "Using device: " << (device.is_cuda() ? "CUDA" : "CPU") << std::
    endl;

        // Initialize models
        xt::models::DCGAN::Generator netG(nz, ngf, nc);
        xt::models::DCGAN::Discriminator netD(nc, ndf);
        netG.to(device);
        netD.to(device);

        // Optimizers
        torch::optim::Adam optimG(netG.parameters(), torch::optim::AdamOptions(lr).
    betas({beta1, 0.999}));
        torch::optim::Adam optimD(netD.parameters(), torch::optim::AdamOptions(lr).
    betas({beta1, 0.999}));

        // Loss function
        torch::nn::BCELoss criterion;

        // Transforms
        std::vector<std::shared_ptr<xt::Module>> transform_list;
        transform_list.push_back(std::make_shared<xt::transforms::image::Resize>(
    image_size));
        transform_list.push_back(std::make_shared<xt::transforms::image::CenterCrop>(
    image_size));
        transform_list.push_back(
            std::make_shared<xt::transforms::general::Normalize>(std::vector<float
    >{0.5, 0.5, 0.5},
                                                                 std::vector<float
    >{0.5, 0.5, 0.5}));

        auto compose = std::make_unique<xt::transforms::Compose>(transform_list);

        // Dataset and DataLoader
        auto dataset = xt::datasets::CelebA("/home/kami/Documents/datasets/",
                                            xt::datasets::DataMode::TRAIN, false,
                                            std::move(compose));

        xt::dataloaders::ExtendedDataLoader data_loader(dataset, batch_size, true, 2,
    /*prefetch_factor=*/2);

        std::cout << "Dataset size: " << dataset.size().value() << std::endl;
        std::cout << "Starting training..." << std::endl;

        // Start timer
        auto start_time = std::chrono::steady_clock::now();

        // Training Loop
        for (int epoch = 0; epoch < num_epochs; ++epoch)
        {
```

 doi:10.20944/preprints202507.0540.v1


```
            int i = 1;
            for (auto& batch : data_loader)
            {
                // (1) Update D network
                netD.zero_grad();
                auto real_data = batch.first.to(device);
                auto current_batch_size = real_data.size(0);

                auto real_label = torch::full({current_batch_size}, 1.0, torch::
    TensorOptions().device(device));
                auto output = torch::sigmoid(netD.forward(real_data)).view(-1);
                auto errD_real = criterion(output, real_label);
                errD_real.backward();

                auto noise = torch::randn({current_batch_size, nz, 1, 1}, torch::
    TensorOptions().device(device));
                auto fake_data = netG.forward(noise);
                auto fake_label = torch::full({current_batch_size}, 0.0, torch::
    TensorOptions().device(device));
                output = torch::sigmoid(netD.forward(fake_data.detach())).view(-1);
                auto errD_fake = criterion(output, fake_label);
                errD_fake.backward();
                auto errD = errD_real + errD_fake;
                optimD.step();

                // (2) Update G network
                netG.zero_grad();
                output = torch::sigmoid(netD.forward(fake_data)).view(-1);
                auto errG = criterion(output, real_label);
                errG.backward();
                optimG.step();

                if (i % 50 == 0) {
                    std::cout << "Epoch [" << epoch + 1 << "/" << num_epochs << "]"
                              << " D_Loss: " << errD.item<float>()
                              << " G_Loss: " << errG.item<float>() << std::endl;
                }
                i++;
            }
        }

        auto end_time = std::chrono::steady_clock::now();
        auto duration_s = std::chrono::duration_cast<std::chrono::seconds>(end_time -
    start_time);

        std::cout << "Training Finished. Total time: " << duration_s.count() << "s." <<
     std::endl;
    }
    catch (const std::exception& e)
    {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Listing 1: XTorch DCGAN Training Implementation

 doi:10.20944/preprints202507.0540.v1

References

1. A. Paszke, et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32*.
2. M. Abadi, et al. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*.
3. A. Radford, L. Metz, & S. Chintala. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint arXiv:1511.06434*.
4. Z. Liu, et al. (2015). Deep Learning Face Attributes in the Wild. *Proceedings of the IEEE International Conference on Computer Vision*.